Graph neural network basics

In this Colab, we are going to introduce some basics of graph neural network (GNN) and build a pipeline for node classification tasks by PyTorch Geometric (PyG). See more introduction about PyG.

Outline

- Basic operation of PyG
- Build a GNN by PyG For Node Classification
- Link Prediction Task by Pyg
- Graph Classification task by Pyg

Basic operation of PyG

In [1]:
```
# import the pytorch library into environment and check its version
import os
import torch
print("Using torch", torch.__version__)
```

```
Using torch 2.5.0+cu121
```

Let's start installing PyG by `pip`. The version of PyG should match the current version of PyTorch. Here we follow the instruction of PyG:

In [2]:
```
!pip install torch-scatter torch-sparse torch-cluster torch-spline-conv torc
!pip install ogb  # for datasets
```

```
Looking in links: https://data.pyg.org/whl/torch-2.0.1+cu118.html
Collecting torch-scatter
  Downloading https://data.pyg.org/whl/torch-2.0.0%2Bcu118/torch_scatter-2.
1.2%2Bpt20cu118-cp310-cp310-linux_x86_64.whl (10.2 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10.2/10.2 MB 38.7 MB/s eta 0:0
0:00
Collecting torch-sparse
  Downloading https://data.pyg.org/whl/torch-2.0.0%2Bcu118/torch_sparse-0.6.
18%2Bpt20cu118-cp310-cp310-linux_x86_64.whl (4.9 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4.9/4.9 MB 74.0 MB/s eta 0:00:
00
Collecting torch-cluster
  Downloading https://data.pyg.org/whl/torch-2.0.0%2Bcu118/torch_cluster-1.
```

```
6.3%2Bpt20cu118-cp310-cp310-linux_x86_64.whl (3.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.3/3.3 MB 70.5 MB/s eta 0:00:
00
Collecting torch-spline-conv
  Downloading https://data.pyg.org/whl/torch-2.0.0%2Bcu118/torch_spline_conv
-1.2.2%2Bpt20cu118-cp310-cp310-linux_x86_64.whl (886 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 886.6/886.6 kB 40.7 MB/s eta
0:00:00
Collecting torch-geometric
  Downloading torch_geometric-2.6.1-py3-none-any.whl.metadata (63 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 63.1/63.1 kB 1.5 MB/s eta 0:0
0:00
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packa
ges (from torch-sparse) (1.13.1)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/dist-pac
kages (from torch-geometric) (3.10.10)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-pack
ages (from torch-geometric) (2024.6.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-pack
ages (from torch-geometric) (3.1.4)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packa
ges (from torch-geometric) (1.26.4)
Requirement already satisfied: psutil>=5.8.0 in /usr/local/lib/python3.10/di
st-packages (from torch-geometric) (5.9.5)
Requirement already satisfied: pyparsing in /usr/local/lib/python3.10/dist-p
ackages (from torch-geometric) (3.2.0)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-pa
ckages (from torch-geometric) (2.32.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packag
es (from torch-geometric) (4.66.5)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/pyt
hon3.10/dist-packages (from aiohttp->torch-geometric) (2.4.3)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.1
0/dist-packages (from aiohttp->torch-geometric) (1.3.1)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/di
st-packages (from aiohttp->torch-geometric) (24.2.0)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.1
0/dist-packages (from aiohttp->torch-geometric) (1.4.1)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python
3.10/dist-packages (from aiohttp->torch-geometric) (6.1.0)
Requirement already satisfied: yarl<2.0,>=1.12.0 in /usr/local/lib/python3.1
0/dist-packages (from aiohttp->torch-geometric) (1.16.0)
Requirement already satisfied: async-timeout<5.0,>=4.0 in /usr/local/lib/pyt
hon3.10/dist-packages (from aiohttp->torch-geometric) (4.0.3)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/
dist-packages (from jinja2->torch-geometric) (3.0.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/py
thon3.10/dist-packages (from requests->torch-geometric) (3.4.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dis
t-packages (from requests->torch-geometric) (3.10)
```

```
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.
10/dist-packages (from requests->torch-geometric) (2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.
10/dist-packages (from requests->torch-geometric) (2024.8.30)
Requirement already satisfied: typing-extensions>=4.1.0 in /usr/local/lib/py
thon3.10/dist-packages (from multidict<7.0,>=4.5->aiohttp->torch-geometric)
(4.12.2)
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.1
0/dist-packages (from yarl<2.0,>=1.12.0->aiohttp->torch-geometric) (0.2.0)
Downloading torch_geometric-2.6.1-py3-none-any.whl (1.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.1/1.1 MB 16.6 MB/s eta 0:00:00
Installing collected packages: torch-spline-conv, torch-scatter, torch-spars
e, torch-cluster, torch-geometric
Successfully installed torch-cluster-1.6.3+pt20cu118 torch-geometric-2.6.1 t
orch-scatter-2.1.2+pt20cu118 torch-sparse-0.6.18+pt20cu118 torch-spline-conv
-1.2.2+pt20cu118
Collecting ogb
  Downloading ogb-1.3.6-py3-none-any.whl.metadata (6.2 kB)
Requirement already satisfied: torch>=1.6.0 in /usr/local/lib/python3.10/dis
t-packages (from ogb) (2.5.0+cu121)
Requirement already satisfied: numpy>=1.16.0 in /usr/local/lib/python3.10/di
st-packages (from ogb) (1.26.4)
Requirement already satisfied: tqdm>=4.29.0 in /usr/local/lib/python3.10/dis
t-packages (from ogb) (4.66.5)
Requirement already satisfied: scikit-learn>=0.20.0 in /usr/local/lib/python
3.10/dist-packages (from ogb) (1.5.2)
Requirement already satisfied: pandas>=0.24.0 in /usr/local/lib/python3.10/d
ist-packages (from ogb) (2.2.2)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist
-packages (from ogb) (1.16.0)
Requirement already satisfied: urllib3>=1.24.0 in /usr/local/lib/python3.10/
dist-packages (from ogb) (2.2.3)
Collecting outdated>=0.2.0 (from ogb)
  Downloading outdated-0.2.2-py2.py3-none-any.whl.metadata (4.7 kB)
Requirement already satisfied: setuptools>=44 in /usr/local/lib/python3.10/d
ist-packages (from outdated>=0.2.0->ogb) (75.1.0)
Collecting littleutils (from outdated>=0.2.0->ogb)
  Downloading littleutils-0.2.4-py3-none-any.whl.metadata (679 bytes)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-pa
ckages (from outdated>=0.2.0->ogb) (2.32.3)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/pyth
on3.10/dist-packages (from pandas>=0.24.0->ogb) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dis
t-packages (from pandas>=0.24.0->ogb) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/d
ist-packages (from pandas>=0.24.0->ogb) (2024.2)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.10/dis
t-packages (from scikit-learn>=0.20.0->ogb) (1.13.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/di
st-packages (from scikit-learn>=0.20.0->ogb) (1.4.2)
```

```
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python
3.10/dist-packages (from scikit-learn>=0.20.0->ogb) (3.5.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-pa
ckages (from torch>=1.6.0->ogb) (3.16.1)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/py
thon3.10/dist-packages (from torch>=1.6.0->ogb) (4.12.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-pa
ckages (from torch>=1.6.0->ogb) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-pack
ages (from torch>=1.6.0->ogb) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-pack
ages (from torch>=1.6.0->ogb) (2024.6.1)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/di
st-packages (from torch>=1.6.0->ogb) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.
10/dist-packages (from sympy==1.13.1->torch>=1.6.0->ogb) (1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/
dist-packages (from jinja2->torch>=1.6.0->ogb) (3.0.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/py
thon3.10/dist-packages (from requests->outdated>=0.2.0->ogb) (3.4.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dis
t-packages (from requests->outdated>=0.2.0->ogb) (3.10)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.
10/dist-packages (from requests->outdated>=0.2.0->ogb) (2024.8.30)
Downloading ogb-1.3.6-py3-none-any.whl (78 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 78.8/78.8 kB 1.9 MB/s eta 0:00:0
0
Downloading outdated-0.2.2-py2.py3-none-any.whl (7.5 kB)
Downloading littleutils-0.2.4-py3-none-any.whl (8.1 kB)
Installing collected packages: littleutils, outdated, ogb
Successfully installed littleutils-0.2.4 ogb-1.3.6 outdated-0.2.2
```

Create a Graph

A single graph in PyG is described by an instance of `torch_geometric.data` which holds the some important attributes by default, like edge_index. We can easily create a graph of various number of edges and nodes by PyG. Take the following graph as an example:

In [3]:
```python
# import torch_geometric.data into environment
from torch_geometric.data import Data
from torch_geometric.datasets import Planetoid
from torch_geometric import nn
import torch_geometric.transforms as T
```

```
/usr/local/lib/python3.10/dist-packages/torch_geometric/typing.py:86: UserWa
rning: An issue occurred while importing 'torch-scatter'. Disabling its usag
e. Stacktrace: /usr/local/lib/python3.10/dist-packages/torch_scatter/_versio
n_cuda.so: undefined symbol: _ZN3c1017RegisterOperatorsD1Ev
  warnings.warn(f"An issue occurred while importing 'torch-scatter'. "
/usr/local/lib/python3.10/dist-packages/torch_geometric/typing.py:97: UserWa
rning: An issue occurred while importing 'torch-cluster'. Disabling its usag
e. Stacktrace: /usr/local/lib/python3.10/dist-packages/torch_cluster/_versio
n_cuda.so: undefined symbol: _ZN3c1017RegisterOperatorsD1Ev
  warnings.warn(f"An issue occurred while importing 'torch-cluster'. "
/usr/local/lib/python3.10/dist-packages/torch_geometric/typing.py:113: UserW
arning: An issue occurred while importing 'torch-spline-conv'. Disabling its
usage. Stacktrace: /usr/local/lib/python3.10/dist-packages/torch_spline_con
v/_version_cuda.so: undefined symbol: _ZN3c1017RegisterOperatorsD1Ev
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torch_geometric/typing.py:124: UserW
arning: An issue occurred while importing 'torch-sparse'. Disabling its usag
e. Stacktrace: /usr/local/lib/python3.10/dist-packages/torch_sparse/_version
_cuda.so: undefined symbol: _ZN3c1017RegisterOperatorsD1Ev
  warnings.warn(f"An issue occurred while importing 'torch-sparse'. "
```

We have 6 edges (undirected graph) and 3 nodes in this graph. So the edge index can be defined as:

In [4]:
```
edge_index = torch.tensor([[0, 1, 1, 2, 0, 2],
                           [1, 0, 2, 1, 2, 0]], dtype=torch.long)
```

Each edge is represented as a tuple (u, v), and that edge_index consists of num_edges columns where each column consists of the two indices u and v corresponding to each edge.

Besides, each node can have a node feature which describes the node's property:

In [5]:
```
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)
```

Then we can define a `Data` object with edge index and node attribute:

In [6]:
```
data = Data(x=x, edge_index=edge_index)
```

`Data` object supports many useful utility functions. For example, we can see the number of the nodes, and whether the graph is a undirected graph:

In [7]:
```
num_nodes = data.num_nodes
print("number of nodes is:", num_nodes)

is_directed = data.is_directed()
```

```
print("graph is directed or not:", is_directed)
```

```
number of nodes is: 3
graph is directed or not: False
```

Question 1 (5 points)

What is the number of the neighbors of node 0 in the graph?

In [8]:
```
def get_n_neighbors(graph, idx):
  # TODO: Implement a function that takes a Data object,
  # an index of a node, and returns the number of the neighbors
  # of this node (as an integer).

  n_neighbors = 0

  ############# Your code here ############
  ## (~1 line of code)
  n_neighbors = (graph.edge_index[0] == idx).sum().item()
  ########################################

  return n_neighbors

idx = 0
n_neighbors = get_n_neighbors(data, idx)
print('Node with index {} has {} neighbors'.format(idx, n_neighbors))
```

```
Node with index 0 has 2 neighbors
```

PyG has a number of graph data with various scales. Cora is one of the most famous dataset in graph learning, and we can use it by PyG:

In [9]:
```
from torch_geometric.datasets import Planetoid

dataset = Planetoid('/tmp/cora', 'cora')
data = dataset[0]
```

```
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.x
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.tx
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.allx
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.y
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.ty
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.ally
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.graph
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.test.index
Processing...
Done!
```

We can see the number of the nodes and edges in cora:

In [10]:
```
num_nodes = data.num_nodes
print('cora has {} nodes'.format(num_nodes))

num_edges = data.num_edges
print('cora has {} edges'.format(num_edges))
```

```
cora has 2708 nodes
cora has 10556 edges
```

Question 2 (10 points)

1. What is the number of the classes in cora dataset?
2. Which node in Cora has the most number of neighbors?

In [11]:
```
def get_num_classes(data):
  # TODO: Implement a function that takes a dataset object
  # and returns the number of classes for that dataset.

  num_classes = 0

  ############# Your code here ############
  ## (~1 line of code)
  num_classes = data.y.max().item() + 1
  ########################################

  return num_classes
```

```
def get_idx_with_most_neighbors(data):
  # TODO: Implement a function that takes a dataset object
  # and returns the index of the node which has the most number of neighbors

  idx = -1

  ############# Your code here ############
  ## (~3 line of code)
  from torch_geometric.utils import degree
  degrees = degree(data.edge_index[0], num_nodes=data.num_nodes)
  idx = degrees.argmax().item()
  ########################################

  return idx

num_classes = get_num_classes(data)
print("cora has {} classes".format(num_classes))

idx = get_idx_with_most_neighbors(data)
print("{} in cora has the most number of neighbors".format(idx))
```

```
cora has 7 classes
1358 in cora has the most number of neighbors
```

In cora, we split the data into train set, validation set and test set by node mask. All the nodes will participate in the message passing process, but we can only assess the train label during training process. This is what we call transductive learning.

In [12]:
```
node_feature = data.x

train_node_feature = node_feature[data.train_mask]
valid_node_feature = node_feature[data.val_mask]
test_node_feature = node_feature[data.test_mask]

print("number of nodes in train set,", train_node_feature.shape[0])
print("number of nodes in valid set,", valid_node_feature.shape[0])
print("number of nodes in test set,", test_node_feature.shape[0])
```

```
number of nodes in train set, 140
number of nodes in valid set, 500
number of nodes in test set, 1000
```

Build a GNN by PyG for Node Classification

In this section we will use PyG to build a classic graph neural network called GCN(Kipf et al. (2017)). Then we will apply this model to handle node classification task in cora. A GCN is built by stacking multiple graph convolution layers `GCNConv` which passes the messages from neighbors to the center node. Here we can define a `GCNConv` by PyG:

In [13]:
```
from torch_geometric.nn import GCNConv

conv = GCNConv(in_channels=1433, out_channels=200, normalize=True)
```

`in_channels` is the dimension of node's input feature, `out_channels` is the dimension of the output representation of node, and `normalize` is whether to add self-loops and compute symmetric normalization on the adjacent matrix. The feature's dimension in cora is 1433, so `in_channels` is set as 1433. We can perform a message passing on cora like this:

In [14]:
```
node_feature = data.x
edge_index = data.edge_index

node_representation = conv(node_feature, edge_index)

print("dimension of node_feature:", node_feature.shape)
print("dimension of node_representation:", node_representation.shape)
```

```
dimension of node_feature: torch.Size([2708, 1433])
dimension of node_representation: torch.Size([2708, 200])
```

We can see that the inputs of `GCNConv` are node feature and edge index. Then the convolution module will perform a message passing like GCN. Recall the MLP we build in colab0. Here we also use `nn.Module` to define a MLP class containing the basic modules of GCN.

Question 3 (5 points)

Following the instruction and build a GCN class using the `GCNConv` modules.

In [15]:
```
from numpy import ERR_DEFAULT
class GCN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super().__init__()

        # TODO: Define two GCNConv modules and a ReLU function.
        # The input size and output size of first GCNConv module should be i
        # The input size and output size of second GCNConv module should be

        ############# Your code here ############
        self.conv1 = GCNConv(in_channels, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, out_channels)
        self.relu = torch.nn.ReLU()
        ########################################
```

```
    def forward(self, node_feature, edge_index):

        output = None

        # TODO: Use the modules you define in __init__ to perform message pa
        # ReLU function should be used in the middle of two GCNConv modules.

        ############# Your code here ############
        output = self.conv1(node_feature, edge_index)
        output = self.relu(output)
        output = self.conv2(output, edge_index)
        ########################################

        return output
```

Training and Testing

Now we can try to construct training and testing pipeline, which is similar to what we do in colab1. First we initialize a GCN model:

In [16]:
```
hidden_channels = 200
num_features = dataset.num_features
num_classes = get_num_classes(data) # please write down the number of classe

model = GCN(num_features, hidden_channels, num_classes)
```

Then we define the optimizer and loss function. Since it is a classification task, we use Cross Entropy Loss:

In [17]:
```
import torch.optim as optim
import torch.nn as nn

optimizer = optim.Adam(model.parameters(), lr=2e-5)
loss_fn = nn.CrossEntropyLoss()
```

Question 4 (10 points)

Please follow the instruction and implement a function that trains a model.

In [18]:
```
def train(model, data, optimizer, loss_fn):

    loss = 0

    # TODO: Define train function.
    # 1. put the model into train mode
    # 2. clear the gradients calculated from the last batch
```

```
    # 3. get the prediction by model
    # 4. calculate the loss between our predictions and the actual labels.
    # Just using nodes in train set!
    # 5. calculate the gradients of each parameter
    # 6. update the parameters by taking an optimizer step

    ############# Your code here ############
    ## (~7 line of code)
    data.to('cuda')
    model.train()
    optimizer.zero_grad()
    output = model(data.x, data.edge_index)
    loss = loss_fn(output[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
    #########################################

    return loss
```

Question 5 (10 points)

Please follow the instruction and implement a function that evaluates a model in train, valid and test sets.

In [19]:
```
@torch.no_grad()
def test(model, data):

    accuracy_list = [0, 0, 0]

    # TODO: Define test function.
    # 1. put the model into eval mode
    # 2. get the prediction by model
    # 3. calculate the accuracy for each set
    # NOTE: the results should be a list containing the accuracy of differer

    ############# Your code here ############
    ## (~5 line of code)
    data.to('cuda')
    model.eval()
    out = model(data.x, data.edge_index)
    pred = out.argmax(dim=1)

    for i, mask in enumerate([data.train_mask, data.val_mask, data.test_mask
        correct = pred[mask] == data.y[mask]
        accuracy = int(correct.sum()) / int(mask.sum())
        accuracy_list[i] = accuracy

    #########################################
```

```
    return accuracy_list
```

We can start to train our model with `train` and `test` functions:

In [22]:
```
hidden_channels = 200
num_features = dataset.num_features
num_classes = get_num_classes(data) # please write down the number of classe

model = GCN(num_features, hidden_channels, num_classes)
model.to('cuda')
optimizer = optim.Adam(model.parameters(), lr=2e-5)
loss_fn = nn.CrossEntropyLoss()

epochs = 50
best_val_acc = final_test_acc = 0
for epoch in range(1, epochs + 1):
    loss = train(model, data, optimizer, loss_fn)
    #print(loss)
    train_acc, val_acc, test_acc = test(model, data)
    #print(train_acc, val_acc, test_acc)
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        final_test_acc = test_acc
print("after {} epochs' training, the best test accuracy is {}".format(epoch
```

```
after 50 epochs' training, the best test accuracy is 0.327
```

It seems that our current hyperparameters, including the choice of optimizer algorithm, learning rate, and the number of training epochs, are not yielding satisfactory performance. To address this issue, we need to fine-tune these parameters for improved results.

One widely adopted approach for hyperparameter tuning involves systematically exploring various options for each hyperparameter. This method, often referred to as hyperparameter optimization, can significantly enhance the performance of our model.

For more in-depth information, you can take a look at the following methods:

1. [

Bayesian Optimization for Hyperparameter Tuning.] (https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf) 2. Grid Search and Random Search.

In [23]:
```
import pandas as pd
import matplotlib.pyplot as plt
```

Please incorporate any values and methods that you believe are worth considering into the following lists.

In [24]:
```
learning_rates = [1e-2, 1e-3]
epochs_list = [2, 4, 6, 8, 10, 25, 50, 75, 100]
optimizers = [optim.Adam, optim.SGD, optim.SGD, optim.RMSprop]  # Add more c
optimizer_names = ['Adam', 'SGD', 'Nesterov', "RMSProp"]  # Match optimizer
```

Next, we establish a Pandas DataFrame schema to store and subsequently visualize the corresponding learning rates based on the input hyperparameters, as follows:

In [25]:
```
columns = ['Learning Rate', 'Epochs', 'Optimizer', 'Final Test Accuracy']
results_df = pd.DataFrame(columns=columns)
```

Question 6 (10 points)

Please complete the for loops below to reset the model at each index, set the hyperparameters, and calculate the resulting accuracy, which will then be recorded in the `results_df` dataframe.

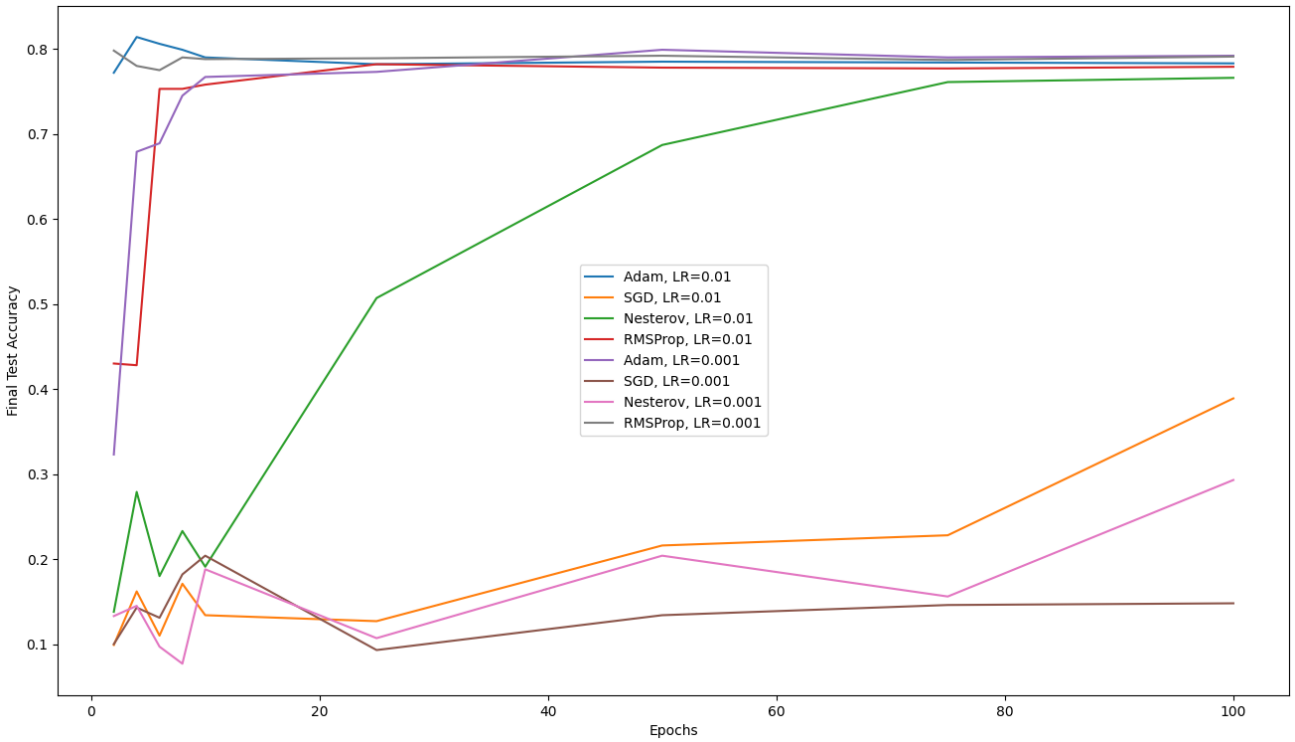
In [26]:
```
for lr in learning_rates:
    for epochs in epochs_list:
        for optimizer, optimizer_name in zip(optimizers, optimizer_names):
            model = GCN(num_features, hidden_channels, num_classes)
            model.to('cuda')
            loss_fn = nn.CrossEntropyLoss()
            best_val_acc = final_test_acc = 0

            ############# Your code here ############
            ## (~7 line of code)

            if optimizer_name == 'Nesterov':
                optimizer = optimizer(model.parameters(), lr=lr, momentum=0.
            else:
                optimizer = optimizer(model.parameters(), lr=lr)


            for epoch in range(1, epochs + 1):
                loss = train(model, data, optimizer, loss_fn)

            train_acc, val_acc, test_acc = test(model, data)
            final_test_acc = test_acc
            ########################################

            # Store the results in the DataFrame
            new_row = pd.DataFrame({'Learning Rate': [lr],
```

```
                                        'Epochs': [epochs],
                                        'Optimizer': [optimizer_name],
                                        'Final Test Accuracy': [final_test_acc]}
                    results_df = pd.concat([results_df, new_row], ignore_index=True)
```

```
<ipython-input-26-3963237f9fd4>:30: FutureWarning: The behavior of DataFrame
concatenation with empty or all-NA entries is deprecated. In a future versio
n, this will no longer exclude empty or all-NA columns when determining the
result dtypes. To retain the old behavior, exclude the relevant entries befo
re the concat operation.
  results_df = pd.concat([results_df, new_row], ignore_index=True)
```

Now, we can visualize the data in `results_df` to identify suitable values for the hyperparameters.

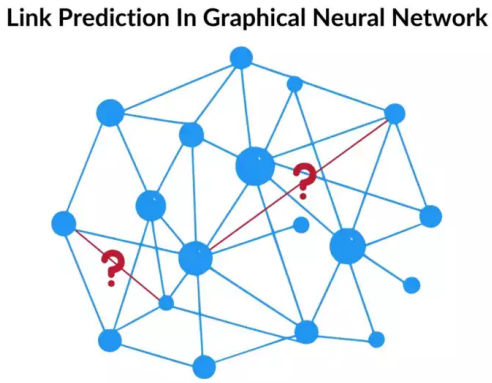
In [27]:
```
plt.figure(figsize=(16, 9))

# Plotting accuracy vs. epochs for each learning rate and optimizer
for lr in learning_rates:
    for opt_name in optimizer_names:
        subset_df = results_df[(results_df['Learning Rate'] == lr) & (result
        plt.plot(subset_df['Epochs'], subset_df['Final Test Accuracy'], labe


plt.xlabel('Epochs')
plt.ylabel('Final Test Accuracy')
plt.legend()
plt.show()
```



Link Prediction

Dataset preprocess

As shown in the following figure, link prediction is to predict whether two nodes in a graph have a link, which can be considered as a binary classification task. We will construct a link prediction dataset containing training, validation, and test set based on Cora.



Given a graph, we divide the initial edge set into three distinct edge sets which represent the training, validation, and test set. Training set and validation set share a same graph structure. Test set contains some edges which does not exist in training and validation set to prevent data leakage.

Our model will be optimized on the training set. We can use `transforms` function in PyG to easily generate the data splits:

In [28]:
```
transform = T.Compose([
    T.RandomLinkSplit(num_val=0.05,  # ratio of edges including in the valid
                      num_test=0.2,  # ratio of edges including in the test
                      is_undirected=True,
                      add_negative_train_samples=False),
])
```

Loading the Cora dataset:

In [29]:
```
dataset = Planetoid('/tmp/cora', 'cora', transform=transform)
```

The data will be transformed from a data object to three tuples, where each element represents the corresponding split:

In [30]:
```
train_data, val_data, test_data = dataset[0]
```

Now data object has two attributes of edge: `edge_index` and `edge_label_index`. `edge_index` denotes the graph structure used for performing message passing in GNN. `edge_label_index` denotes the edge index used to calculate loss in training set, or to evaluate the model in validation and test set.

Printing the statistics of data:

In [31]:
```
print("Number of the nodes in training, validation and test data are", train
print("Number of the edges in training, validation and test data are", train
print("Number of the edge_label_index in training, validation and test data
```

```
Number of the nodes in training, validation and test data are 2708 2708 2708
Number of the edges in training, validation and test data are 7920 7920 8446
Number of the edge_label_index in training, validation and test data are 396
0 526 2110
```

Pipeline

We use the same GCN you constrcted.

In [32]:
```
model = GCN(dataset.num_features, hidden_channels=128, out_channels=64)
```

In [33]:
```
optimizer = torch.optim.Adam(params=model.parameters(), lr=0.01)
```

Similar as the what we do in the node classification task, we first apply the GCN model to produce the representation of each node in the graph. Usually we will use **inner product** to measure the similarity between two node representations to determine how likely it is for these two nodes to be connected.

Question 7 (5 points)

Following the instruction and implement the function to calculate the inner product:

In [34]:
```
def compute_similarity(node_embs, edge_index):
    result = 0

    # TODO: Define similarity function.
```

```
    # 1. calculate the inner product between all the pairs in the edge_index
    # Note: the shape of node_embs is [n, h] where n is the number of nodes,
    # the shape of edge_index is [2, m] where m is the number of edges

    ############## Your code here ############
    ## (~1 line of code)
    result = (node_embs[edge_index[0]] * node_embs[edge_index[1]]).sum(dim=1
    ##########################################

    return result

n, h = 5, 10  # number of nodes and embedding size
node_embs = torch.rand(n, h)
edge_index = torch.tensor([[0, 1, 2, 3],
                           [2, 3, 0, 1]])  # compute the similarity of (0, 2
similarity = compute_similarity(node_embs, edge_index)
print("Similairty:", similarity)
```

```
Similairty: tensor([2.5138, 1.6712, 2.5138, 1.6712])
```

We optimize the model by minimizing the loss function. Here we consider the link prediction task as a binary classification task (edge exists or no), and apply binary cross entropy loss:

In [35]:
```
loss_fn = torch.nn.BCEWithLogitsLoss()
```

The edges in the graph will be taken as the positive examples with label=1 in the loss function. To prevent model from collapse, we usually will feed some **negative examples** to the loss function, which is the non-existing edges in the graph. The number of negative examples should equal to the number of positive ones.

With the help of PyG, we can easily perform the negative sampling. Here is an example:

In [36]:
```
from torch_geometric.utils import negative_sampling

neg_edge_index = negative_sampling(
      edge_index=train_data.edge_index,  # positive edges in the graph
      num_nodes=train_data.num_nodes,  # number of nodes
      num_neg_samples=5,  # number of negative examples
    )

print("shape of neg_edge_index:", neg_edge_index.shape)  # [2, num_neg_sampl
print("negative examples:", neg_edge_index)
```

```
shape of neg_edge_index: torch.Size([2, 5])
negative examples: tensor([[ 514, 1803,  352,   27, 2292],
        [2647,   85, 1254, 1165,  736]])
```

Positive examples (`edge_label_index`) will be assigned the label 1, and negative ones will be assigned the label 0. We can obtain the label of positive examples like this:

In [37]:
```
print("positive examples' labels:", train_data.edge_label)
```

```
positive examples' labels: tensor([1., 1., 1.,  ..., 1., 1., 1.])
```

Now we can construct training and testing pipeline.

Question 8 (15 points)

Please follow the instruction and implement a function that trains a model.

In [38]:
```
def train(model, data, optimizer, loss_fn):

    loss = 0

    # TODO: Define train function.
    # 1. put the model into train mode
    # 2. clear the gradients calculated from the last batch
    # 3. use 'edge_index' to get the node representation by model
    # 4. sample the negative examples with the same number of positive ones
    # 5. concatenate the positive edges and negative edges
    # 6. concatenate the labels of positive edges and negative edges
    # 7. calculate the similarity between two end nodes to determine the pro
    # 8. feed the probability and edge label to the loss function
    # 9. calculate the gradients of each parameter
    # 10. update the parameters by taking an optimizer step

    ############## Your code here ############
    ## (~10 line of code)

    model.train()
    optimizer.zero_grad()
    embeddings = model(data.x, data.edge_index)
    num_positive = data.edge_label_index.size(1)

    neg_samples = negative_sampling(
        edge_index=data.edge_index,
        num_nodes=data.num_nodes,
        num_neg_samples=num_positive
    )

    all_edge_indices = torch.cat([data.edge_label_index, neg_samples], dim=1

    positive_labels = data.edge_label
    negative_labels = torch.zeros(num_positive, device=data.edge_label.devic
    combined_labels = torch.cat([positive_labels, negative_labels])
```

```
    src_nodes, dst_nodes = all_edge_indices
    src_embeddings = embeddings[src_nodes]
    dst_embeddings = embeddings[dst_nodes]
    logits = (src_embeddings * dst_embeddings).sum(dim=1)

    loss = loss_fn(logits, combined_labels)

    loss.backward()
    optimizer.step()

    ########################################

    return loss
```

We usually use AUC score to evaluate the performance of model on binary classification task. The test function is as followed:

In [39]:
```
from sklearn.metrics import roc_auc_score

@torch.no_grad()
def test(model, data):
    model.eval()
    out = model(data.x, data.edge_index)  # use `edge_index` to perform mess
    out = compute_similarity(out, data.edge_label_index).view(-1).sigmoid()
    return roc_auc_score(data.edge_label.cpu().numpy(), out.cpu().numpy())
```

Now we can start to train our model based on `train` and `test` function:

In [40]:
```
epochs = 50

best_val_auc = final_test_auc = 0
for epoch in range(1, epochs + 1):
    loss = train(model, train_data, optimizer, loss_fn)
    valid_auc = test(model, val_data)
    test_auc = test(model, test_data)
    if valid_auc > best_val_auc:
        best_val_auc = valid_auc
        final_test_auc = test_auc
    print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}, Val: {valid_auc:.4f}, Test
```

```
Epoch: 001, Loss: 0.6647, Val: 0.7692, Test: 0.7704
Epoch: 002, Loss: 0.9383, Val: 0.7152, Test: 0.7087
Epoch: 003, Loss: 1.0388, Val: 0.7850, Test: 0.7695
Epoch: 004, Loss: 0.6566, Val: 0.8561, Test: 0.8287
Epoch: 005, Loss: 0.6251, Val: 0.8442, Test: 0.8170
Epoch: 006, Loss: 0.6332, Val: 0.8261, Test: 0.8079
Epoch: 007, Loss: 0.6363, Val: 0.8154, Test: 0.8095
Epoch: 008, Loss: 0.6370, Val: 0.8125, Test: 0.8170
Epoch: 009, Loss: 0.6339, Val: 0.8150, Test: 0.8282
Epoch: 010, Loss: 0.6289, Val: 0.8238, Test: 0.8423
Epoch: 011, Loss: 0.6191, Val: 0.8346, Test: 0.8554
Epoch: 012, Loss: 0.6089, Val: 0.8437, Test: 0.8653
Epoch: 013, Loss: 0.5984, Val: 0.8511, Test: 0.8732
Epoch: 014, Loss: 0.5836, Val: 0.8582, Test: 0.8805
Epoch: 015, Loss: 0.5659, Val: 0.8623, Test: 0.8868
Epoch: 016, Loss: 0.5525, Val: 0.8647, Test: 0.8914
Epoch: 017, Loss: 0.5302, Val: 0.8658, Test: 0.8940
Epoch: 018, Loss: 0.5129, Val: 0.8663, Test: 0.8952
Epoch: 019, Loss: 0.4979, Val: 0.8672, Test: 0.8970
Epoch: 020, Loss: 0.4913, Val: 0.8699, Test: 0.8992
Epoch: 021, Loss: 0.4750, Val: 0.8722, Test: 0.9016
Epoch: 022, Loss: 0.4645, Val: 0.8745, Test: 0.9041
Epoch: 023, Loss: 0.4646, Val: 0.8770, Test: 0.9066
Epoch: 024, Loss: 0.4654, Val: 0.8802, Test: 0.9095
Epoch: 025, Loss: 0.4610, Val: 0.8836, Test: 0.9122
Epoch: 026, Loss: 0.4514, Val: 0.8868, Test: 0.9140
Epoch: 027, Loss: 0.4475, Val: 0.8887, Test: 0.9153
Epoch: 028, Loss: 0.4582, Val: 0.8905, Test: 0.9161
Epoch: 029, Loss: 0.4504, Val: 0.8925, Test: 0.9165
Epoch: 030, Loss: 0.4473, Val: 0.8940, Test: 0.9165
Epoch: 031, Loss: 0.4461, Val: 0.8952, Test: 0.9163
Epoch: 032, Loss: 0.4446, Val: 0.8961, Test: 0.9163
Epoch: 033, Loss: 0.4416, Val: 0.8973, Test: 0.9165
Epoch: 034, Loss: 0.4397, Val: 0.8976, Test: 0.9167
Epoch: 035, Loss: 0.4441, Val: 0.8981, Test: 0.9165
Epoch: 036, Loss: 0.4342, Val: 0.8990, Test: 0.9163
Epoch: 037, Loss: 0.4292, Val: 0.8999, Test: 0.9160
Epoch: 038, Loss: 0.4278, Val: 0.9001, Test: 0.9162
Epoch: 039, Loss: 0.4338, Val: 0.8996, Test: 0.9166
Epoch: 040, Loss: 0.4277, Val: 0.8984, Test: 0.9172
Epoch: 041, Loss: 0.4267, Val: 0.8972, Test: 0.9180
Epoch: 042, Loss: 0.4165, Val: 0.8964, Test: 0.9189
Epoch: 043, Loss: 0.4209, Val: 0.8958, Test: 0.9196
Epoch: 044, Loss: 0.4203, Val: 0.8951, Test: 0.9201
Epoch: 045, Loss: 0.4207, Val: 0.8945, Test: 0.9203
Epoch: 046, Loss: 0.4196, Val: 0.8941, Test: 0.9198
Epoch: 047, Loss: 0.4171, Val: 0.8936, Test: 0.9193
Epoch: 048, Loss: 0.4260, Val: 0.8922, Test: 0.9189
Epoch: 049, Loss: 0.4151, Val: 0.8915, Test: 0.9187
Epoch: 050, Loss: 0.4275, Val: 0.8909, Test: 0.9185
```

Graph classification task

Now let's have a closer look at the task of graph classification. Graph classification refers to the problem of classifiying entire graphs, given a dataset of graphs. Here, we will apply GNN to embed entire graphs.

Dataset preprocess

One of the most common benchmark dataset of graph classification is TUDatasets which are collected by TU Dortmund University. Each graph in this dataset is a molecule, and the task is to infer whether a molecule inhibits HIV virus replication or not. We can load this dataset by PyG. In this colab, we mainly focus on one of the smaller ones in TUDatasets: MUTAG dataset.

In [4]:
```
from torch_geometric.datasets import TUDataset

dataset = TUDataset(root='/tmp/mutag', name='MUTAG')
print(dataset)
```

```
Downloading https://www.chrsmrrs.com/graphkerneldatasets/MUTAG.zip
Processing...
MUTAG(188)
Done!
```

We can obtain its number of graphs, classes, node features:

In [5]:
```
print(f'number of graphs: {len(dataset)}')
print(f'number of classes: {dataset.num_classes}')
print(f'Number of node features: {dataset.num_node_features}')
```

```
number of graphs: 188
number of classes: 2
Number of node features: 7
```

There 188 graphs in this dataset, and we can get the graph object with any id. For example:

In [6]:
```
data = dataset[5]
print(f'5-th graph object: {data}')
```

```
5-th graph object: Data(edge_index=[2, 62], x=[28, 7], edge_attr=[62, 4], y=[1])
```

We can obtain some statistics for each graph object:

In [7]:
```
print(f'Number of nodes: {data.num_nodes}')
print(f'Number of edges: {data.num_edges}')
print(f'Average node degree: {data.num_edges / data.num_nodes:.2f}')
print(f'Has isolated nodes: {data.has_isolated_nodes()}')
print(f'Has self-loops: {data.has_self_loops()}')
print(f'Is undirected: {data.is_undirected()}')
```

```
Number of nodes: 28
Number of edges: 62
Average node degree: 2.21
Has isolated nodes: False
Has self-loops: False
Is undirected: True
```

We will divide the dataset into training set and test set, and there is no duplicate graph in these two sets. We can randomly pick 150 graphs to form training set, and the remaining ones will be the test set:

In [8]:
```
dataset = dataset.shuffle()

train_dataset = dataset[:150]
test_dataset = dataset[150:]
```

Mini-batching of graphs

To fully utilize GPU, we will conduct mini-batch training which can be achieved by PyG. A batch of graphs will be grouped in a giant graph that holds multiple isolated subgraphs, and node features are simply concatenated. `dataloader` object in PyG can easily finish the aboved process:

In [9]:
```
from torch_geometric.loader import DataLoader

train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)
```

Here is an example to show how dataloader works. We can observe that multiple graphs are included in a giant graph.

In [10]:
```
for step, data in enumerate(train_loader):
    print(f'Step {step + 1}, number of graphs in the current batch: {data.nu
    print(f'Step {step + 1}, number of nodes in the current batch: {data.num
    print(f'Step {step + 1}, the graph id to which each node belongs is: {da
    print()
```

```
Step 1, number of graphs in the current batch: 128
Step 1, number of nodes in the current batch: 2298
```

```
Step 1, the graph id to which each node belongs is: tensor([  0,   0,   0,
..., 127, 127, 127])

Step 2, number of graphs in the current batch: 22
Step 2, number of nodes in the current batch: 390
Step 2, the graph id to which each node belongs is: tensor([ 0,  0,  0,  0,
0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,
2,
        2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,
3,
        3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  4,  4,
4,
        4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,
5,
        5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  6,
6,
        6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,
7,
        7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,
7,
        7,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,
9,
        9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10,
10,
       10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11,
11,
       11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 12, 12,
12,
       12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
12,
       12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13,
13,
       13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14,
14,
       14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 15, 15,
15,
       15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 16, 16, 16, 16, 16, 16,
16,
       16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17,
17,
       17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18,
19,
       19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19,
19,
       20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21,
21,
       21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21])
```

The graph id of every node to which it belongs is indicated by the `batch` attribute.

Model Implementation

First we perform message passing to embed each node in the graph, then aggregate the node embeddings into a graph embedding by pooling method. Finally the graph embedding will be fed to a classifier to conduct graph classification.

We will apply mean pooling method which is to simply take the average of node embeddings. Here is an example of mean pooling:

In [11]:
```
from torch_geometric.nn import global_mean_pool

x = torch.rand(5, 4)  # embeddings of 5 nodes

# graph id. The first two nodes belong to first graph,
# the 3rd node belongs to the second graph,
# and the last two nodes belong to the last graph
batch = torch.tensor([0, 0, 1, 2, 2])

x = global_mean_pool(x, batch)  # node embedding and the graph id to which e
print(f"shape of graph embedding: {x.shape}")
```

```
shape of graph embedding: torch.Size([3, 4])
```

Question 9 (15 points)

Follow the instructions and implement the GNN model for graph classifiation task.

In [12]:
```
import torch.nn as nn
from torch.nn import Linear
from torch_geometric.nn import GCNConv
from torch_geometric.nn import global_mean_pool


class GCN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super(GCN, self).__init__()

        # TODO: Define two GCNConv modules, a linear classifier and a ReLU f
        # The input size and output size of first GCNConv module should be i
        # The input size and output size of second GCNConv module should be
        # The input size and output size of Linear module should be hidden_c

        ############## Your code here ############
        ## (~4 line of code)
        self.conv1 = GCNConv(in_channels, hidden_channels)
```

```
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.linear = Linear(hidden_channels, out_channels)
        self.relu = nn.ReLU()

        ########################################

    def forward(self, x, edge_index, batch):

        output = None

        # TODO: Use the modules you define in __init__ to perform message pa
        # ReLU function should be used in the middle of two GCNConv modules.
        # Apply global_mean_pool module to generate graph embeddings
        # Apply linear classifier to predict the label

        ############# Your code here ############
        ## (~3 line of code)

        x = self.conv1(x, edge_index)
        x = self.relu(x)
        x = self.conv2(x, edge_index)
        output = self.linear(global_mean_pool(x, batch))

        ########################################
        return output
```

Initialize the model and the optimizer:

In [13]:
```
model = GCN(in_channels=dataset.num_node_features, hidden_channels=64, out_c
model = model.to('cuda')
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)
print(model)
```

```
GCN(
  (conv1): GCNConv(7, 64)
  (conv2): GCNConv(64, 64)
  (linear): Linear(in_features=64, out_features=2, bias=True)
  (relu): ReLU()
)
```

Pipeline

Here we use cross entropy loss to optimize:

In [14]:
```
loss_func = torch.nn.CrossEntropyLoss()
```

Question 10 (15 points)

Now we are going to implement `train` function. Please follow the instrution:

In [15]:
```
def train(model, loader, optimizer, loss_func):

    loss = 0

    # TODO: Define train function.
    # 1. put the model into train mode
    # 2. iterate over the dataloader
    # 3. obtain the predicted result by model
    # 4. compute the loss
    # 5. loss backward
    # 6. update the parameters by taking an optimizer step
    # 7. clear the gradients calculated from the last batch

    ############## Your code here ############
    ## (~7 line of code)
    model.train()
    for data in loader:
        data = data.to('cuda')
        optimizer.zero_grad()
        out = model(data.x, data.edge_index, data.batch)
        loss = loss_func(out, data.y)
        loss.backward()
        optimizer.step()



    ##########################################

    return model
```

The `test` function is implemented as followed:

In [16]:
```
def test(model, loader):
     model.eval()

     correct = 0
     for data in loader:  # Iterate in batches over the training/test datase
         data = data.to('cuda')
         out = model(data.x, data.edge_index, data.batch)
         pred = out.argmax(dim=1)  # Use the class with highest probability.
         correct += int((pred == data.y).sum())  # Check against ground-trut
     return correct / len(loader.dataset)  # Derive ratio of correct predict
```

Now we can train and evaluate our model on graph classification task:

In [17]:
```
epochs = 100
```

```
for epoch in range(1, epochs):
    model = train(model, train_loader, optimizer, loss_func)
    test_acc = test(model, test_loader)
    print(f'Epoch: {epoch:03d}, Test Acc: {test_acc:.4f}')
```

```
Epoch: 001, Test Acc: 0.3421
Epoch: 002, Test Acc: 0.6579
Epoch: 003, Test Acc: 0.6579
Epoch: 004, Test Acc: 0.6579
Epoch: 005, Test Acc: 0.6579
Epoch: 006, Test Acc: 0.7105
Epoch: 007, Test Acc: 0.7895
Epoch: 008, Test Acc: 0.7105
Epoch: 009, Test Acc: 0.7632
Epoch: 010, Test Acc: 0.5263
Epoch: 011, Test Acc: 0.6842
Epoch: 012, Test Acc: 0.7632
Epoch: 013, Test Acc: 0.7368
Epoch: 014, Test Acc: 0.7105
Epoch: 015, Test Acc: 0.7895
Epoch: 016, Test Acc: 0.7632
Epoch: 017, Test Acc: 0.7895
Epoch: 018, Test Acc: 0.7632
Epoch: 019, Test Acc: 0.7368
Epoch: 020, Test Acc: 0.7368
Epoch: 021, Test Acc: 0.7632
Epoch: 022, Test Acc: 0.7632
Epoch: 023, Test Acc: 0.7632
Epoch: 024, Test Acc: 0.6842
Epoch: 025, Test Acc: 0.7632
Epoch: 026, Test Acc: 0.7895
Epoch: 027, Test Acc: 0.7632
Epoch: 028, Test Acc: 0.7632
Epoch: 029, Test Acc: 0.7895
Epoch: 030, Test Acc: 0.7632
Epoch: 031, Test Acc: 0.7895
Epoch: 032, Test Acc: 0.6842
Epoch: 033, Test Acc: 0.7632
Epoch: 034, Test Acc: 0.7632
Epoch: 035, Test Acc: 0.7368
Epoch: 036, Test Acc: 0.5789
Epoch: 037, Test Acc: 0.7632
Epoch: 038, Test Acc: 0.6842
Epoch: 039, Test Acc: 0.7632
Epoch: 040, Test Acc: 0.7632
Epoch: 041, Test Acc: 0.7632
Epoch: 042, Test Acc: 0.7632
Epoch: 043, Test Acc: 0.7632
Epoch: 044, Test Acc: 0.7895
```

```
Epoch: 045, Test Acc: 0.7105
Epoch: 046, Test Acc: 0.7632
Epoch: 047, Test Acc: 0.7632
Epoch: 048, Test Acc: 0.7895
Epoch: 049, Test Acc: 0.7105
Epoch: 050, Test Acc: 0.7632
Epoch: 051, Test Acc: 0.7632
Epoch: 052, Test Acc: 0.7632
Epoch: 053, Test Acc: 0.7632
Epoch: 054, Test Acc: 0.7632
Epoch: 055, Test Acc: 0.7632
Epoch: 056, Test Acc: 0.7632
Epoch: 057, Test Acc: 0.7368
Epoch: 058, Test Acc: 0.7895
Epoch: 059, Test Acc: 0.7895
Epoch: 060, Test Acc: 0.7632
Epoch: 061, Test Acc: 0.7632
Epoch: 062, Test Acc: 0.7632
Epoch: 063, Test Acc: 0.7895
Epoch: 064, Test Acc: 0.7632
Epoch: 065, Test Acc: 0.7632
Epoch: 066, Test Acc: 0.7632
Epoch: 067, Test Acc: 0.7105
Epoch: 068, Test Acc: 0.7632
Epoch: 069, Test Acc: 0.7105
Epoch: 070, Test Acc: 0.7632
Epoch: 071, Test Acc: 0.7632
Epoch: 072, Test Acc: 0.7632
Epoch: 073, Test Acc: 0.7895
Epoch: 074, Test Acc: 0.7368
Epoch: 075, Test Acc: 0.7632
Epoch: 076, Test Acc: 0.7632
Epoch: 077, Test Acc: 0.7368
Epoch: 078, Test Acc: 0.7632
Epoch: 079, Test Acc: 0.7632
Epoch: 080, Test Acc: 0.7632
Epoch: 081, Test Acc: 0.7105
Epoch: 082, Test Acc: 0.7895
Epoch: 083, Test Acc: 0.7895
Epoch: 084, Test Acc: 0.7632
Epoch: 085, Test Acc: 0.7632
Epoch: 086, Test Acc: 0.7105
Epoch: 087, Test Acc: 0.7632
Epoch: 088, Test Acc: 0.7632
Epoch: 089, Test Acc: 0.7632
Epoch: 090, Test Acc: 0.7895
Epoch: 091, Test Acc: 0.7632
Epoch: 092, Test Acc: 0.7105
Epoch: 093, Test Acc: 0.7895
Epoch: 094, Test Acc: 0.7632
```

```
Epoch: 095, Test Acc: 0.7632
Epoch: 096, Test Acc: 0.7632
Epoch: 097, Test Acc: 0.7632
Epoch: 098, Test Acc: 0.7105
Epoch: 099, Test Acc: 0.7895
```

Submission

Make sure to run all the cells and save a copy of this colab in your driver. If you complete this notebook, download the colab and upload your work to canvas to submit it.

In [17]: